**Рip C:\Users\Nike\AppData\Local\Programs\Python\Python311**

**Типы данных:**

int - 123

float - 1.23

str - "abc1"

bool - True или False

list - ["asd", 123, 1.2]

set - {12, "asd"}

tuple - (123, "asd", True)

dict - { key : value }

**Основные матиматические функции:**

+ Сложение // Деление на цело

- Вычитание % Остаток от деления

\* Умножение = Присваивание

/ Деление == Равно

\*\* Степень != или not = Не равно

> Больше and И

< Меньше or ИЛИ

**Методы строк:**

.max() - Возвращает самое большое значение

.min() - Возвращает самое маленькое значение

.abs() - Переводит -1 в 1

.len() - Считает количество элементов

in - В

is - Типо in, но сложнее

.capitalize() - ааа - Ааа

.swapcase() - аАа - АаА

.title() - аа аа - Аа Аа

.lower() - ААА - ааа

.upper() - ааа - ААА

.count() - Считает одинаковые символы

.replace("Заменяет это", "на это", сколько раз)

.find() - Как далеко символ от начала

.rfind() - От конца

**Модуль Math:**

round(x) - округление sin(x) e - Число Эйлера

sqrt(x) - корень cos(x)

pow(x, n) = x \*\* n tg(x)

log(x) ctg(x)

factorial(x) pi = 3,14

**Методы списков**

.append() - добавляет элемент в конец списка

a.extend(b) - слияние списков а и b

del[5] - удаляет 6-эл списка

.split() - разбивает строку в слова и переносит в список

.join() - наоборот

a.pop() - удаляет указанный эл из списка

.count() - считает сколько раз значение находится в списке

.clear() - очищает список

.copy() - копирует список

.sort() - сортирует список

**Заполнение списка:**

[ 0 for i in range(10)]

то чем то куда сколько раз

**Важно**

a = list(map(int, b))

**Индексация**

0123456789

ели мясо м

...-3-2-1

**Mодуль Random**

from random import (\* - импортирует все)

модуль методы

.randint(от , до) - возвращает целое псевдослучайное число в промежутке

.randrange(1, 10) - тоже самое с циклом

.random - числа от 0 до 1 пример 0.8291379

.seed - настройка рандома не по сис времени

.uniform - не целые числа

choice(a) - вытаскивает элемент из списка or строки

.shuffle - перемешивает список

**Dict - словарь**

содание dict(m = 246, p =811)

методы:

.fromkeys(seq, value) - создает словарь из ключей seq и значения value

.get(key, ...) - возвращает значение ключей

.items - возвращает пары зн ключ

.keys - возвращает значение клчей

.pop(key, ...) - удаляет ключ

.setdefault(key, ...) - возвращает значение ключа, если его нет создает

.values - возвращает значение

\_ = dict(zip(key, value)) - распаковка

**Set - множество**

add - добавляет элемент

update - добавляет больше одного элемента

remove - удаляет элемент

pop - удаляет случайный элемент

union - объеденяет множество

intersection - перечисление set

difference - разность set

**Модуль Pillow**

в cmd

pip3 instal pillow

Есть 3 свойства изображения

format ("Jpeg")

size (1920, 1080)

mode "RGB"

встраиваем

from PIL import Image

**методы:**

Image.open(a) - открываем изоображение в системе чтобы работать с ним а - это путь

.show() - открывает изображение на экране

print(a) - выводит информацию о фото

a = Image.new("RGB", (400, 500), (255,0,0)) - создает изоображение

RGB - формат (400, 500) - размер (255,0,0) - цвет

\_.rotate(angle = 90, expand = True, fillcolor = (255,255,5), center = (100, 100)) - Вращение изображения

angle - угол поворота expand - изменение высоты и ширины fillcolor - заливка пустого пространства center - точка поворота

\_.save("C:\\\_\\Sky.jpg") - сохранение изображения

Указываем путь, название и разрешение, при необходимости используем r"\_\_"

.crop(box = (left, upper, right, lower)) - обрезка изображений

первые два аргумента - левая верхняя точка, другие два - правая нижняя.

**Работа с файлами**

a = open("C:\\\_.txt", mode = "O", encoding = "UTF-8") - открытие в программе текстового файла

O - режим открытия их несколько:

t - говорит программе что открывает простой текст

b - говорит что открываем бинарный текст

"rt" - для чтения файла

"wt" - для записи файла (но все данные из него очищает)

"at" - для записи файла в конце с сохранение данных

"+" - добавляем + и открываем в двойном режеми и для чтения и для записи rt+ wt+ ...

"xt" - создание файла

encoding - позволяет редактировать разные раскалдки текста, по стандарту там инглиш, UTF-8 эт русский.

После изменения файла должны его закрывать

file1.close()

file1.encoding - проверяет кадировку файла

file1.closed - проверяет закрыть ли файл

file1.read - читает содержимое файла и выводит его

file1.readlibe() - тоже самое только первую строку

file1.readlines() - читает все и возвращает список срок

file1.write("Hello!!!") - запись в файл

file1.tell() - Возвр. значение файловой позиции но в кодировке utf-8 он возвращает в байтах поэтому в уме делим на 2

file1.seek(a,b) - перемещаем указатель

a - индекс умноженный на два так как кодировка в битах

b - режим указателя есть 3. 0 - с начала, 1 - с середины, 2 - с конца - Лучше не юзать, ничего не понятно

Чтобы удалить строку - это гемор, мы сначало создаем переменую в которой читаем все строки в виде спика,

потом через del[] удаляем строку из списка и заново записываем в файл

///Если мы пробуем открыть ткст которого нет он его создаст///

with open('example.txt') as (название переменой в которой откроется файл):

используется для автоматического закрытия файла без гемора с исключениями

**Исключения**

try:

блок кода где всплывает ошибка.

except (вид ошибки):

блок кода который вополняется после ошибки.

else:

добавочный блок кода есле ошибки не было выполняется после трай.

finally:

блок кода который выполняется в конце в любом случае. К примеру закрыть файл.

**os**

import os

os.getcwd() - выводит текущую директорию

os.mkdir(rf"C:\Users\-\_-\Desktop\{Название папки}") - создает папку

os.chdir() - выполняет вход в папку

os.chdir("..") - такая конструкция позволяет вернутся в начальную папку

os.makedirs(".../One/Two/Three") - создает вложенные папки

os.rename("из этого.txt", "в этот.txt") - переименование файла

os.replace("от куда.txt", "куда.txt") - перемещение файла

os.listdir() - Возвращает список с названием всех файлов в директории и их расширением

os.remove("путь") - удаляем файл

os.rmdit("путь") - удаляем пустую папку

os.removedirs("путь") - удаляем папку в которой что-то есть

os.stat("путь") - выводит информацию о файле

**Time**

from time import \* - тогда не придеся прописывать time.

import time - тогда придется

time.ctime() - Выводит время настоящее

time.sleep(...) - откладывает время исполнения кода на кол-во указанных в аргументе секунд, просто функция, пишется внизу

res = localtime() - выводит что-то из этого списка, но нужно указать это как аргумент внутри скобок

(tm\_year=2023, tm\_mon=8, tm\_mday=9, tm\_hour=11, tm\_min=53, tm\_sec=1, tm\_wday=2, tm\_yday=221, tm\_isdst=0)

mktime() как localtime() - обратная

asctime() как localtime() - возвращает строку

**ООП**

класс это - кот

у него есть параметры

порода, имя, возраст.

Объекты это уже конкретные коты, со своей породой, к примеру:

Русская - порода

Рыжик - имя

2 - возраст

class может соержать не только данные но и свойства и методы.

**Инкапсуляция:**

Важна лишь работа класса а не внутренний алгоритм как он работает и тд.

Чтобы пользователь не мог сломать класс то данные и методы класса т.е основную работу класса

можно скрывать. И класс становится единым целым.

**Наследование:**

Позволяет вынести за скобки ощее из всех классов.

Типо если у классов есть общие свойства. То обязатьльно их выносить в другой класс с этими свойствами

И после этого мы должны унаследовать у нашего нового класса, все свойства, для других классов с

которых мы их брали.

**Полиморфизм:**

Можно через единый интерфейс работать с объектами разных классов

выделяют два полиморфизма.

Ad hoc и параметрический.

Ad hoc - старый и в пайтоне не используется.

Параметрический - Создан для упрощения работы, чтобы не прорисовывать объекты, а оперировать

разными типами объектов через их дочерний класс.

Переменные внутри класса приято называть атрибутами

class Point:

color = "red"

circle = 2

так объявляют класс, внутри него атрибуты.

чтобы вывести инфу о атрибуте класса пишем

print(Point.color)

Возвращает red

чтобы вывести все атрибуты класса пишем

Point.\_\_dict\_\_ - их ам будет много но и наши заданные тоже будут присутствовать.

можно менять атрибуты класса

Point.color = "blue"

Создать объекты класса легко

a = Point()

мы переменной а присвоили все атрибуты класса Point

и так же их можем вызвать у а через print(a.color)

но если мы поменяем атрибуты у класса а, у Point останутся такие какие были.

А если у поинт поменяем то и объекта он тоже сменится.

можно проверить тип объектов класса type(a)

выведет <class '\_\_main\_\_.Point'>

Можно вне класса или объекта создать атрибут

прописав:

a.(имя нового атрибута) = "ASD"

Но это неверно лучше делать тоже самое через команду

setattr(Point, "новый атрибут", "его значение")

тем же самым способом можно менять значения атрибутов.

удалять атрибуты класса или объекта можно так

del a.Name

a - объект

Name - имя атрибута

hasattr(Point, "Prop") - проверяет есть ли атрибут у класса.

если есть вернет True, нет вернет False

Можно и лучше удалять атрибуты командой

delattr(Point, "сам атрибут")

МЕТОДЫ это функции внутри класса, отвечающие за его дейтсвия и объявляются вот так

def name():

блок кода

вызывать метод так же как и данные

Point.name\_mehod() - если без круглых скобок, он вернет что это функция

class Go(self) - self - позволяет передовать атрибуты класса в объекты

инициализатор (конструктор) - \_\_init\_\_ - позволяет быстро собирать объект с новыми атрибутами.

def \_\_init\_\_(self, Value = 0, Trump = 0, Suit = ""):

self.Value = Value

self.Trump = Trump

self.Suit = Suit

go = Cards(1, 2, "xyz")

все методы \_\_name\_\_ - называются магическими

ы

финализатор - \_\_del\_\_ - закрывает не используемые атрибуты чтобы не жрать лишнию память.

def \_\_del\_\_(self):

pass

метод \_\_new\_\_ - вызывается до создания объектов.

def \_\_new\_\_(cls, \*args, \*\*kwargs):

блок кода выполняющийся до создания объектов

return super().\_\_new\_\_(cls)

cls = тот же self но у \_\_new\_\_ - и ссылается он на дочерний класс.

\*args, \*\*kwargs - то где хранятся все атрибуты

return super().\_\_new\_\_(cls) - чтобы у нас метод заработал, ему нужно вернуть значение через дочерний класс super()

вот этой строчкой.

Декораторы ооп

@classmethod @staticmethod

class Point:

Min = 0

Max = 100

@classmethod

def validate(cls, arg):

if cls.Min < arg < cls.Max:

return(True)

else:

return(False)

@classmethod - обращается только к атрибутам внутри класса не иначне. Вся работа происходит через cls.

и после использования этого метода, обращатся к функциям с cls, можно через сам класс а не создавать новые

оъекты.

@staticmethod

def norm2(x, y):

return x + y

Go = Point(1, 3)

print(Go.norm2(5, 6))

@staticmethod - методы не принадлежащие к атрибутам и классам, используются для упрощения работы с кодом.

Скрытые параметры типа self, cls - не указываются

**Замыкания Декораторы**

Итак замыкания

def say\_name(name):

    def say\_good\_bye():

        print("Goodbye", name)

    return say\_good\_bye

f = say\_name("Витя")

f()

В чем смысл

Мы создали функцию внутри которой объявили другую функции

И записали в переменой f ссылку на дочернюю функцию и в теории

При обычном вызывании в таком виде, у нас бы выскочила ошибка так как пайтон почистил бы все не нужные переменные после вызывания функции sey\_name так как работает что при вызове через ссылку f мы возвращаем функцию say\_good\_bey она же должна удалятся, прикол в том что внутренняя функция не очищается пока существует переменная f которая обращается к внутренней функции которая в свою очередь к дочерней и та которая к ссылке на супер пупер внутреннею невидимую функцию в самом пайтоне которая возвращает f, т.е. мы сделали вот такой вот замкнутый круг.

Замыкание в Python - это функция, которая запоминает значения переменных из окружающего ее контекста, даже если эти переменные уже не доступны.

Для лучшего понимания, представьте, что у вас есть функция, которая возвращает другую функцию. Возвращаемая функция может использовать переменные, объявленные внутри родительской функции, даже после того, как родительская функция завершается и переменные перестают быть доступными. Вот именно это и называется замыканием.

Декораторы это те функции которые при сложении с другими функциями расширяют их функционал.

def decorator(func):

    def good():

        print("ДО")

        func()

        print("После")

    return good

def hello():

    print("Hello world")

f = decorator(hello)

f()

Вывод:

ДО

Hello world

После

Крч декораторы без замыкания работать не будут так как внутренние условия бы не выполнялись

Тут мы увеличили функцию hello с помощью функции decorator с замыканием

Но правильнее не создавать новую перемнную а присваивать старой функции новые возможности нашего декоратора

hello = decorator(hello)

hello()

а не

f = decorator(hello)

f()

для правильного использования параметров других функций от декоратора, чтобы не было ошибок прописываем несколько параметров в наш декоратор которые будут обозначать что на вход, других функций мы сможе обрщатся к их парметрам без проблем

def decorator(func):

    def good(\*args, \*\*kwargs):

        print("ДО")

        func(\*args, \*\*kwargs)

        print("После")

    return good

def hello(Go):

    print(f"Hello world {Go}")

hello = decorator(hello)

hello("ЯЯЯЯ")

ну и последняя дороботка если мы хотим что-то возвращать внутри нашей новой функции в декораторе прописываем так

def decorator(func):

    def good(\*args, \*\*kwargs):

        print("ДО")

        res = func(\*args, \*\*kwargs)

        print("После")

        return res

    return good

и последнее чтобы декорировать функцию мы сверху нее пишем @и нейм декоратора

def decorator(func):

    def good(\*args, \*\*kwargs):

        print("ДО")

        res = func(\*args, \*\*kwargs)

        print("После")

        return res

    return good

@decorator

def hello(Go):

    print(f"Hello world {Go}")

hello("ЯЯЯЯ")

вот так уоот

**Потоки**

Процесс – это наша сама программа.

Потоки – это части нашей программы.

Потоки выполняются последовательно из-за ограничения gil.

Процессы выполняться независимо друг от друга.

Крч процессы новые использовать для скорости, но они жрут много. А потоки наооборот.

Особо учить не буду. Но как открыть новый поток запишу, вот собсно код таймера

import time

import threading

can\_stop = 0

def start():

    global can\_stop

    print("Таймер запущен ")

    timer\_sec = 0

    timer\_min = 0

    timer\_hours = 0

    while can\_stop == 0:

        time.sleep(1)

        timer\_sec += 1

        if timer\_sec > 59:

            timer\_sec = 0

            timer\_min += 1

            if timer\_min > 59:

                timer\_min = 0

                timer\_hours += 1

    print(f"{timer\_sec} сек {timer\_min} мин {timer\_hours} час")

def stop():

    global can\_stop

    try:

        can\_stop = int(input("Напишите 1 для остановки: "))

    except ValueError:

        can\_stop = 1

        print("Надо было ввести 1!!!")

thread = threading.Thread(target=stop)

thread.start()

start()

импортируем threading()

создаем функцию, которая будет работать в отдельном потоке. Тупо создаем объект класса и в таргет передаем нейм функции

thread = threading.Thread(target=stop)

ну и потом логично ее вызываем

thread.start()

главное не проебаться, они же тоже работают в процедурном стиле сверху вниз. Закрывать их кстати не надо, после закрытия основного потока они сами обрубаются, удобно однако…

**TKINTER – GUI**

Импортируем ткинтер import tkinter можем изменть обращение к модулю as tk, но это не обяз. Сам модуль стандартный питухоновский.

Начнем с создания нашего окна, это объект класса tkinetr

window = tkinter.Tk()

Изменим название окна

window.title("ОКНО!!!")

Размер

window.geometry("400x400")

цвет

window.configure(background="#E9967A")

Хорошие методы…

Создадим надпись

label\_1 = tkinter.Label(window, text="Добро пожаловать")

параметры 1 – наш объект класса(окно), и 2 - вывод текста.

label\_1.configure(foreground="white", background="black")

Как и с окном можем настроить цвет фона bg и цвет текста fg.

Дальше ее нужно распаковать в наше окно.

label\_1.pack()

он сам настраивает размеры и тд.

Создание кнопки, все тоже самое кроме command = name\_func

button\_1 = tkinter.Button(window, text= "ТЫК", command= button\_1\_click)

button\_1.pack()

button\_1.configure(width=10, height=2)

Это изменяет размер кнопки.

Это обработчик, у нас есть функция и при нажатии на кнопку эта функция выполняется. Смысл прост…

Заканчиваем двумя вот этими командами

window.resizable(False, False)

window.mainloop()

Первая запрещает пользователю редактировать размер окна, вторая вечный цыкл логично.

.place(x=0, y=200, width=399, height=50)

Редактирование положения и размера кнопки или лейбла, нужно понимать что place используется вместо pack()

font='Times 30' – измение шрифта

wraplength=200

этим методом мы переносим текст в лейбле чтобы он не вылазил за рамки

Чтобы обработать выбор combox используем бинд

combox\_mode.bind("<<ComboboxSelected>>", com\_mode)

внутри указываем какуе-то дичь, и функцию которая будет обрабатывать значения вот собсно функция сейчас ее разберем

def com\_mode(event):

        global com\_mode\_rus

        global com\_mode\_eng

        values = combox\_mode.get()

        if values == "rus":

            com\_mode\_rus = True

            com\_mode\_eng = False

        elif values == "eng":

            com\_mode\_eng = True

            com\_mode\_rus = False

        else:

            com\_mode\_rus = True

            com\_mode\_eng = False

Обязательный параметр event он вроде не используеться но должен существовать, и через get получаем значение комбокса

**PyQt - QTDisigner**

Крч эту gui который позволяет графически заполнять и расставлять кнопки.

Qt Disigner приложение для граф заполнения gui.

И модуль pip install pyqt5 потом будет позволять для кнопок писать обработчики и тд.

Изначально импорт этих классов

from PyQt5 import QtWidgets

from PyQt5.QtWidgets import QApplication, QMainWindow

первый позволяет создавать объекты.

Application – позволяет создать само приложение.

QMainWindow – позволяет создать окно.

import sys

и сис заодно.

def application():

    app = QApplication(sys.argv)

    window = QMainWindow()

    window.setWindowTitle("Программа")

    window.setGeometry(300, 250, 350, 200)

    window.show()

    sys.exit(app.exec\_())

application()

Сначала нужно создать базовую функцию нашего окна.

Создать объект класса QApplication(тут вернуть стандартные настройки нашего пк sys.atgv)

Теперь нужно создать окно класс QMainWindow()

    window.setWindowTitle("Программа")

служит для названия окна программы

    window.setGeometry(300, 250, 350, 200)

для геометрии окна

первые два это смещение по x и y с левого верхнего угла, вторые два это размеры окна, тоже по x и y.

Потом это окно нужно показывать методом show

    sys.exit(app.exec\_())

указываем корректные параметры выхода из приложения.

main\_text = QtWidgets.QLabel(window)

    main\_text.setText("Надпись хахаха")

    main\_text.move(100, 100)

    main\_text.adjustSize()

Добавление объекта label

main\_text = QtWidgets.QLabel(window)

параметром передаем наше окно

    main\_text.move(100, 100)

    main\_text.adjustSize()

это смещение а последний это авто настройка под окно, чтобы не было неправильных смещений.

 btn = QtWidgets.QPushButton(window)

    btn.move(60, 60)

    btn.setText("Нажми на меня")

    btn.setFixedWidth(200)

Создание кнопки вместо стандартной расстановки adjustSize мы использовали конкретно FixedWidth и настроили x200.

def add\_label():

    print("add")

    btn.clicked.connect(add\_label)

обработчик нажатия.

Но есть косяк с отдельной функцией обработчика мы не можем взаимодействовать с объектами окна, поэтому сейчас будут переписывать прогу и писать класс.

class Window(QMainWindow):

    def \_\_init\_\_(self):

        super(Window, self).\_\_init\_\_()

        self.setWindowTitle("Программа")

        self.setGeometry(300, 250, 350, 200)

        self.main\_text = QtWidgets.QLabel(self)

        self.main\_text.setText("Надпись хахаха")

        self.main\_text.move(100, 100)

        self.main\_text.adjustSize()

        self.btn = QtWidgets.QPushButton(self)

        self.btn.move(60, 60)

        self.btn.setText("Нажми на меня")

        self.btn.setFixedWidth(200)

        self.btn.clicked.connect(self.add\_label)

    def add\_label(self):

        print("qwe")

def application():

    app = QApplication(sys.argv)

    window = Window()

    window.show()

    sys.exit(app.exec\_())

application()

Теперь код выглядит так.

Все аргументы махнули на self, и для новых объектов self.

Разобрались теперь погнали работать с дизайнером.

Перевод файлов с расш ui в py, через cmd.

Указываем папку с файлом, cd потом вот это calc\_gui.ui это имя

pyuic5 calc\_gui.ui -o .py -x